AFRL-RI-RS-TR-2014-110

ENABLING NEXT-GENERATION MULTICORE PLATFORMS IN EMBEDDED APPLICATIONS

THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL

APRIL 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2014-110 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILLIAM E. McKEEVER, JR.
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2014	FINAL TECHNICAL REPORT	NOV 2010 – NOV 2013

4. TITLE AND SUBTITLE

ENABLING NEXT-GENERATION MULTICORE PLATFORMS IN EMBEDDED APPLICATIONS

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-11-1-0033

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

James H. Anderson

5d. PROJECT NUMBER
T2CS

5e. TASK NUMBER
UN

5f. WORK UNIT NUMBER
CM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Computer Science
The University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27599-3175

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-110

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

When checking the timing correctness of a system of real-time tasks, upper bounds on the execution times of individual tasks are required. On a multicore platform, execution-time bounds that are not overly pessimistic are difficult to determine. A key stumbling block in this regard is the difficulty of predicting when memory references will hit in on-chip caches. While this is a problem even on uniprocessors, the presence of shared caches on multicore platforms further exacerbates this problem. Such caches, which are widely used in current commercially available multicore machines, can be accessed concurrently by tasks on different cores. This creates cross-core cache interactions that are difficult (if not impossible) to predict. In fact, such difficulties are one of the main reasons why multicore platforms are not in widespread use in safety-critical domains such as avionics systems. In this project, a new shared cache management framework has been developed that enables more predictable shared cache behavior. In this report, an overview of this framework is presented and the results of an evaluation of it are discussed

15. SUBJECT TERMS

real-time, multicore, scheduling, resource allocation, shared caches, page coloring, mixed-criticality

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM E. McKEEVER JR**
U	U	U	UU	34	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

1 SUMMARY

When checking the timing correctness of a system of *real-time* tasks, upper bounds on the execution times of individual tasks are required. On a multicore platform, execution-time bounds that are not overly pessimistic are difficult to determine. A key stumbling block in this regard is the difficulty of predicting when memory references will hit in on-chip caches. While this is a problem even on uniprocessors, the presence of *shared caches* on multicore platforms further exacerbates this problem. Such caches, which are widely used in current commercially available multicore machines, can be accessed concurrently by tasks on different cores. This creates cross-core cache interactions that are difficult (if not impossible) to predict. In fact, such difficulties are one of the main reasons why multicore platforms are not in widespread use in safety-critical domains such as avionics systems. In this project, a new shared cache management framework has been developed that enables more predictable shared cache behavior. In this report, an overview of this framework is presented and the results of an evaluation of it are discussed.

2 INTRODUCTION

Multicore platforms offer the potential of enabling computationally intensive workloads in many settings, with less size, weight, and power (SWaP) consumption. Such settings range from hand-held and embedded devices, to laptop and desktop systems, to the world's fastest supercomputers. In all of these settings, the computational capabilities of multicore chips are being leveraged to realize a wealth of new products and services across many application domains. One domain, however, stands out as being largely unaffected: *safety-critical* cyber-physical embedded systems.

Examples of such systems include avionics and automotive systems, medical systems for diagnosis and treatment, and smart robotic systems that function in environments where failures cannot be tolerated or are difficult to correct (*e.g.*, planetary rovers). A common characteristic of these and other safety-critical systems is that failures may have catastrophic consequences, such as loss of life or serious financial repercussions. Because of the high cost of failure, safety-critical systems must be *certified* (often by governmental or international bodies) before being deployed.

Certification can be expensive and time-consuming. Thus, it is imperative that safety-critical systems be built using certification-friendly hardware platforms and design processes. One of the most important tenets in this regard is that computations should be *predictable*. Predictability ensures that behaviors arising during certification reflect those that will be seen in the deployed system. Predictability is also fundamental when establishing real-time correctness.

The importance of predictability in certification explains why multicore platforms are not in widespread use in safety-critical domains. In such platforms, different cores share hardware components such as caches and memory controllers. Using current technology, very pessimistic assumptions must be made regarding the utilization of these shared resources during certification. The processing capacity lost to such pessimism can easily negate the impact of any additional cores. The resulting state of affairs is unsettling: the multicore revolution is enabling dramatically better functionality and services in many domains, but safety-critical cyber-physical systems are excluded. Unless the "predictability problem" associated with multicore platforms are addressed, functional advances in such systems will continue to be impeded.

Focus of this project. In this project, we have conducted research on this "predictability problem" with a particular emphasis on the subproblem of ensuring *predictable shared cache behavior on multicore platforms*. Ensuring such behavior is not only important from the standpoint of real-time predictability, it is a key first step towards ensuring safe and secure isolation of disparate system components in safety-critical systems. Much of our research agenda has been shaped by a particularly compelling driving problem: supporting real-time functionality in future unmanned air vehicles (UAVs). Such UAVs will have greater autonomous capabilities than current designs. Some of our work on this driving problem has been done in collaboration with colleagues at Northrop Grumman Corp. (NGC) [29, 46]. In avionics systems such as UAVs, different software components may have different criticalities. The most important outcome of this project is a new real-time shared cache management framework that enables more predictable cache behavior while taking criticality information into account. Additionally, various other problems related to multicore real-time resource allocation were addressed.

Organization. In the rest of this report, we elaborate on the UAV driving problem and our contributions. We begin by discussing our methods, assumptions, and procedures in Section 3. We then present an overview of our main results in Section 4. We conclude in Section 5.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section, we describe our methods, assumptions, and procedures. We begin by providing necessary background on the UAV driving problem and on real-time resource allocation issues.

3.1 Driving Problem: Future UAVs

Complex adaptive systems (*CASs*) have been identified by Air Force scientists as key components of the Air Force of the future [2, 17]. A typical CAS can provide a broader array of capabilities than a traditional counterpart while also exhibiting far more autonomy, thus providing a force-multiplying effect. Oftentimes, this additional autonomy not only replaces human pilots or operators, but actually provides superior capabilities. Examples of CASs include next-generation UAVs, advanced radar systems, and fractionated satellites.

Key attributes of CASs include: a large number of on- and off-board inputs (*e.g.*, embedded sensors); heavy reliance on the fusion of data from multiple sources; feedback-driven decision-making and learning; and various mechanisms to enable adaptation [17]. Such systems can be said to have a "near-infinite" number of possible states and to be "fundamentally nondeterministic." They exhibit a high degree of self-awareness and incorporate high-level reasoning in order to maximize their capabilities.

Next-generation UAVs: The UCAS-D vision. A canonical "archetype" of a future Air Force CAS, exemplifying all the key attributes discussed above, can be found in the U.S. Navy's *Unmanned Combat Air System* (*N-UCAS*) program (which actually originated as a joint Air Force-Navy program, J-UCAS). In the long-term vision espoused under this program, a system of stealthy UAVs with miscellaneous capabilities (including strike, surveillance, reconnaissance, and radar jamming) could be deployed over enemy airspace for extended periods of time. These

vehicles are envisioned to employ sophisticated software that enables them to exhibit on-board intelligence that allows them to break down and pursue high-level objectives identified by a small team of mission operators. Further, autonomously formed ad hoc teams of vehicles exhibiting fractionated capabilities (*e.g.*, one vehicle may contain extra munitions, while another contains special jamming equipment) are envisioned to pursue complex, theater-level objectives. The software to be run on these vehicles needs to incorporate complicated high-level reasoning that takes into account theater-level information and that provides for coordination of the overall system of vehicles. It also needs to be able to autonomously and quickly react to incoming hostile forces.

The need for real-time multicore support. Practically all CASs will include sophisticated timing constraints. Thus, their design, development, verification and validation, and certification must not only account for logical correctness, but also temporal correctness. While methods for ensuring logical correctness are clearly deserving of significant attention, our research has focused purely on ensuring temporal correctness. Many CASs, including next-generation UAVs, will also have both significant computational workloads and stringent SWaP requirements. These factors suggest utilizing a multicore-based implementation. However, ensuring the temporal correctness of software typically requires sacrificing a significant fraction of the underlying hardware platform's capacity; on a multicore platform, this loss can greatly exceed 50% using current resource allocation and analysis techniques. To describe the nature of such capacity loss, we must first more carefully consider how real-time workloads are scheduled and provisioned.

3.2 Multiprocessor Real-Time Scheduling

When designing a real-time application, the goal is to produce a *schedulable system*, *i.e.*, one whose timing constraints can be guaranteed. Timing constraints are usually expressed using deadlines. To illustrate, consider the well-studied *periodic task model* [42]. In this model, the specified system is comprised of a collection of recurring sequential tasks, $T_1, \ldots, T_n$, which are to be scheduled on m processors. Each such task T_i releases a succession of *jobs* $J_{i,1}, J_{i,2}, \ldots$ and is defined by specifying a *period* p_i and a *worst-case execution time (WCET)* e_i. Successive jobs of T_i are released p_i time units apart, and one that is released a time t has a *deadline* at time $t + p_i$. Also, each such job executes for at most e_i time units and its required processing time should be allocated between its release and deadline.

A periodic task system may be scheduled by either an event- or timer-driven scheduler. An *event-driven* scheduler uses priorities (*e.g.*, job deadlines) to make online scheduling decisions. A *timer-driven* scheduler uses a pre-computed dispatching table. The discussion below mostly focuses on event-driven schedulers.

Real-time schedulability. Under the periodic model, the definition of the term "schedulable" depends on whether deadlines are hard or soft (we consider both possibilities). In a *hard real-time* (*HRT*) system, deadlines can never be missed, while in a *soft real-time* (*SRT*) system, deadline misses are sometimes tolerable. SRT schedulability can be defined in different ways; we assume here that a SRT system is schedulable if deadline tardiness is bounded (such bounds would be expected to be reasonably small). In determining schedulability (hard or soft), the processor share required by each task T_i is of importance. The share required by T_i is given by the quantity $u_i =$

e_i/p_i, which is called its *utilization* or *weight*.

Algorithms for testing schedulability are necessarily dependent on the scheduling algorithm being used. Since our research targets multicore platforms, multiprocessor scheduling algorithms are our primary focus. Such an algorithm may follow a *partitioned* approach (tasks are statically assigned to processors) or a *global scheduling* approach (any task may execute on any processor). An example of a partitioned algorithm is *partitioned EDF* (P-EDF), which uses the uniprocessor *earliest-deadline-first* (EDF) algorithm as the per-processor scheduler. (Under EDF, jobs with earlier deadlines have higher priority.) An example of a global algorithm is *global EDF* (G-EDF), under which tasks are EDF-scheduled using a single priority queue. In addition to "pure" partitioned and global algorithms, hybrid approaches are possible. One such approach is *clustered scheduling*, under which tasks are partitioned onto clusters of processors, with those in each cluster being globally scheduled within that cluster [5, 15].

Capacity loss in multicore systems. When assessing schedulability on a multiprocessor, it is often necessary to constrain per-task utilizations or overall utilization in some way. Such constraints result in *capacity loss*, *i.e.*, unavailable processing capacity. For example, under partitioned scheduling, if each task has utilization $0.5 + \varepsilon$, then only m tasks (of total utilization approximately $m/2$) can be scheduled on m processors because only one task can be assigned to a given processor without overloading it. In effect, half of the system's capacity can be "lost." Despite this limitation of partitioned scheduling, generally speaking, partitioned algorithms result in less capacity loss for HRT workloads [10], and global or clustered algorithms result in less capacity loss for SRT workloads [18, 33].

The nature of the capacity loss just described is *scheduler-related*. Additionally, capacity loss can arise due to system overheads. Such overheads include activities by the operating system (OS) that take away processor time from the scheduled tasks. *OS-related overheads* include processor time lost to processing interrupts, making scheduling decisions, enacting context switches, *etc.* These overheads tend to be reasonably small in practice—overhead values of a few microseconds to a few tens of microseconds are common [10]. In contrast, *cache-related overheads* also must be accounted for, and these can be larger, on the order of tens to hundreds of microseconds or even a few milliseconds [10]. Such overheads reflect the additional cost incurred to reload previously cached instructions and data when a job resumes execution after a preemption or migration. The impacts of such overheads can be ameliorated by limiting preemptions or migrations, but this can cause higher-priority jobs to experience *priority inversions*, *i.e.*, they may be temporally blocked by non-preemptive lower-priority jobs. In practice, all of the various overheads discussed here must be accounted for in schedulability analysis. Broadly speaking, this involves increasing each task's WCET by an amount that reflects the worst-case overhead one of its jobs may experience.

On multicore platforms, the presence of shared hardware such as caches, memory controllers, buses, *etc.*, can cause additional capacity loss. In particular, when task WCETs are determined offline, assumptions regarding contention for such resources must be made. For highly critical HRT tasks, such assumptions will tend to be very pessimistic. For example, when analyzing such a task's code, if a data reference cannot be proved with certainty to result in a cache hit, then it will be assumed that such a reference goes to memory. With complex code, it can be very difficult to determine which memory references will hit in some cache. As a result, provisioned task WCETs may be much larger than the worst case that can be seen practice. The goal of our work on shared cache management is *to enable shared caches on multicore platforms to be more predictably*

managed so that lower WCETs for tasks can be provisioned.

Mixed-criticality scheduling. We developed our shared cache management framework atop a mixed-criticality scheduler developed with colleagues from NGC under AFOSR support [29, 46]. We provide a brief introduction to mixed-criticality scheduling here.

In many embedded systems, the severity of failure is not the same for all tasks in the system. For example, the failure of one task may cause loss of life, while the failure of a different task may only cause degraded system performance. Such tasks are said to be of differing *criticalities*. The task model defined at the beginning of this subsection is agnostic with respect to criticalities: under it, each task, regardless of criticality, is provisioned by specifying a pessimistic WCET. A system provisioned in this way may be reasonably well-utilized from a validation and certification perspective, *i.e.*, at *design time*, but be severely underutilized in practice, *i.e.*, at *run time*.

A technique for reclaiming this spare capacity has been proposed by Vestal (who was working in the avionics industry at Honeywell at the time) [55]. He observed that, from the perspective of scheduling a less critical task, the execution times assumed of more critical tasks are needlessly pessimistic. Thus, he proposed that schedulability tests for less critical tasks be altered to incorporate less pessimistic execution times for more critical tasks—that is, per-criticality-level execution times are assumed, with lesser pessimism for lower levels.

More formally, in a system with L criticality levels, ***L system variants are analyzed***: in the level-l variant, level-l execution times are assumed for *all* tasks. The degree of pessimism in determining such execution times is level-dependent: if level l is of higher criticality than level l', then level-l execution times will generally be greater than level-l' execution times. For example, when certifying a system at the highest criticality level, *provably correct* upper bounds on execution times might be assumed (as computed, for example, using *timing analysis tools* [59, 60]), while when certifying at a lower level, *observed* worst-case times from profiling might be assumed. The task model resulting from Vestal's work has come to be known as the *mixed-criticality task model*. Assessing correctness in a criticality-cognizant way as described here can greatly reduce capacity loss, particularly when relatively few higher-criticality tasks exist, as is often the case in avionics systems.

With the above discussion of background information in place, we now describe our methods, assumptions, and procedures. We organize this description as follows: first, we more carefully describe the cache management problem to be solved; second, we describe the techniques we developed as solutions to this problem; third, we describe how such techniques were implemented.

3.3 Cache Management Problem

The aforementioned mixed-criticality scheduler devised by us and colleagues at NGC is called MC^2 (mixed-criticality on multicore) [29, 46]. In this document, we consider the problem of adding proper shared cache management to MC^2 when supporting periodic tasks.

In MC^2, four criticality levels exist, denoted A (highest) through D (lowest), as shown in

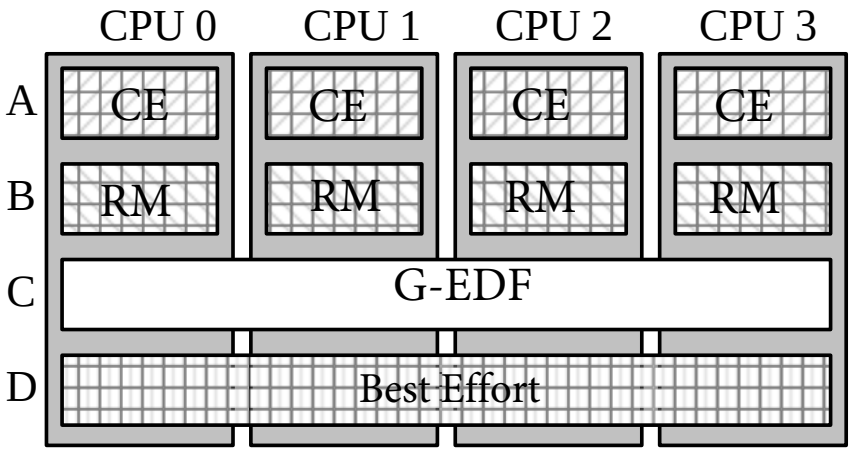


Figure 1: Scheduling under MC^2 on a four-processor system.

Figure 1. Higher criticality tasks are statically prioritized over lower criticality ones. Level-A tasks are partitioned and scheduled on each processor using a table-driven cyclic executive [4]. Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each processor. Level-A and -B tasks are required to be simply periodic (all tasks commence execution at time 0 and periods are harmonic). Level-C tasks are scheduled via a G-EDF scheduler. Level-D tasks are scheduled with no real-time guarantees on a best-effort basis (so we do not consider them further). A task's execution time at its own criticality level is treated as an OS enforced execution budget: if a job of a task T_i has an execution time exceeding T_i's budget, then more than one budget allocation will be required to service it. Level-A and -B tasks are HRT, while level-C tasks are SRT. Most interesting cache-related issues are exposed by focusing only on levels B and C (one HRT level and one SRT level). In the current prototype of our cache management framework, only these two levels are assumed to be present, so we make this assumption hereafter. We note that systems with two criticality levels have been the predominate focus of prior work on mixed-criticality scheduling [6, 7, 19].

Page coloring. All of our proposed cache management schemes utilize page coloring in some way. We now describe page coloring with respect to an NVIDIA Tegra T30 quad-core ARM Cortex A9 system, which is the machine used in our implementation-oriented work.

The ARM platform used in our experiments has four cores that share an L2 cache. In this document, we consider page coloring with respect to this cache. The L2 cache on this platform is a 1 MB 8-way set associative cache: it stores contents of physical memory in 32 B units called "lines," each line of physical memory maps to a particular cache "set," each such set can store 8 lines (equivalently, there are eight "ways" per set), and in total there are 2^{12} sets. The physical memory of this platform is subdivided into 4 KB pages. To envision the coloring process, consider each page in sequence. For the first page in memory, assign the color "0" to it, and assign

the same color to the cache lines to which its contents map. Then, since each page consists of 4 KB/(32 B/line) = 128 lines, sets 1 − 128 are assigned color 0. Repeat this process, assigning color 1 (mapping to sets 129 − 256) to the second page in memory, color 2 (sets 257 − 384) to the third page, and so on. Then, after the 32^{nd} page, all 2^{12} sets will have been used and color assignments will "wrap," *i.e.*, the 33^{rd} page will map to the same cache sets as the first, so we reuse color 0 for it. Continuing this process, each page will be assigned to one of 32 colors. Moreover, two pages that are assigned different colors will map to different cache sets and thus cannot conflict with each other in the cache.

In Section 3.4, we consider techniques that exploit page coloring to eliminate or control cache conflicts. In discussing these techniques, we limit attention to non-shared task data pages, as only these pages are managed in the prototype system described in Section 3.6. We define the *working set size* (*WSS*) of a task to be the size (in bytes) of the set of data pages it may access in one job, *i.e.*, the size of its per-job *working set* (*WS*).

Multiprocessor real-time locking. Some of the cache management schemes we consider utilize multiprocessor real-time locking protocols. In the protocols we consider, tasks wait by *suspending* execution. Locking protocols must ensure that *priority inversion blocking* (*pi-blocking*) can be analytically bounded. Pi-blocking is the duration of time a job is blocked while a lower-priority job is running. Per-task bounds on pi-blocking are required when analyzing schedulability. We let b_i denote the pi-blocking bound for task T_i.

On a multiprocessor system, the actual definition of pi-blocking depends on how schedulability analysis is done [10]. For some schedulers, suspensions are notoriously difficult to analyze, so *working set size*) analysis is applied: jobs may suspend, but each e_i must be *analytically* inflated by b_i prior to applying a schedulablity test to account for lock-related delays. We utilize s-oblivious analysis in our work. Some of the nuances of such analysis can best be explained by comparing it to *suspension-aware* (*s-aware*) analysis, which explicitly accounts for b_i and is available for some schedulers.

Since suspended jobs are counted as demand under s-oblivious analysis, the mere *presence* of *m* higher-priority jobs rule out a priority inversion, whereas only *ready* higher-priority jobs can nullify a priority inversion under s-aware analysis. Accordingly, under **s-oblivious** (respectively, **s-aware**) schedulability analysis, a job J_i incurs *s-oblivious* (respectively, *s-aware*) *pi-blocking* at time t if J_i is pending but not scheduled and fewer than *m* higher-priority jobs are **pending** (respectively, **ready**). This is illustrated in Figure 2. Prior research has shown that s-aware and s-oblivious analysis are comparable in terms of schedulability achievable in practice [10].

Cache-related locking problem. We now describe the basic synchronization problem that arises when using locking protocols for cache management (protocol-specific details are discussed in Section 3.4). When using such protocols, each color is viewed as a shared resource that has a number of "replicas" as given by the number of cache ways, as illustrated in Figure 3. Before a job commences execution, it must first lock a replica of each color that it requires (as given by the pages it will access). If the job accesses *r* pages with the same color, then it must lock *r* replicas of that color. The needed synchronization protocol must enable a set of shared resources

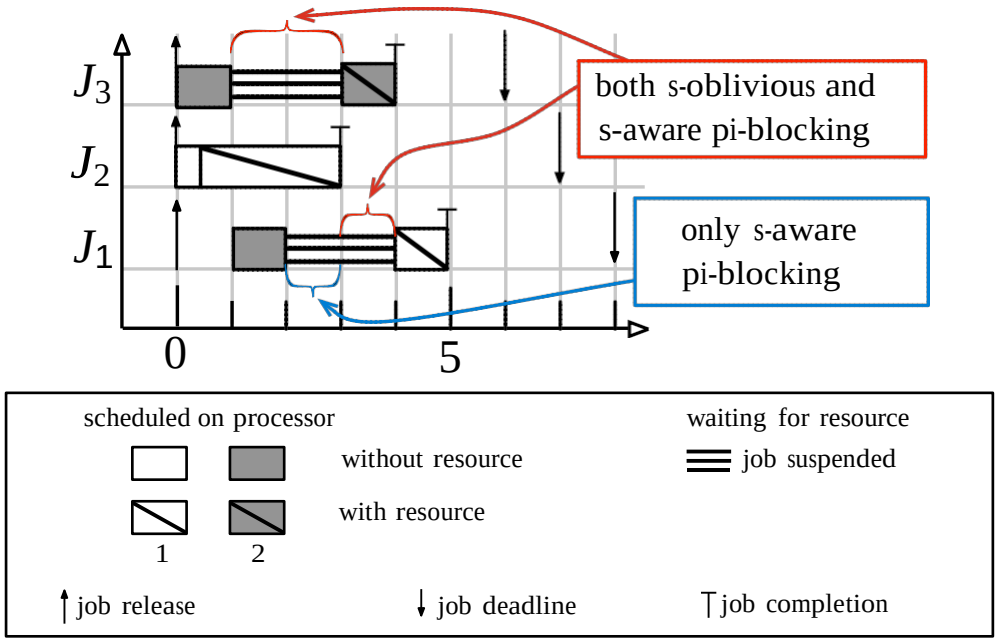


Figure 2: Example (from [10]) of s-oblivious and s-aware pi-blocking. G-EDF scheduling on two processors is assumed. J_3 suspends during [1, 3), and since no higher-priority jobs exist it is pi-blocked under either definition. J_1, suspended during [2, 4), suffers pi-blocking under either definition during [3, 4) since it is among the *m* highest-priority pending jobs, but only s-aware pi-blocking during [2, 3) as J_3 is pending but not ready then.

to be managed, where each resource has multiple replicas, and jobs may need to lock several replicas simultaneously. In actuality, such a protocol is utilized by the OS when making scheduling decisions, *i.e.*, the jobs themselves do not acquire and release color-related locks. This means that the OS must know the pages a job will access prior to making a scheduling decision.

3.4 Cache Management Techniques

In this section, we present techniques for managing accesses to a shared cache by level-B tasks in MC^2 (recall our assumption that level A is not present). Our goal is to enable WCET reductions and greater predictability in such tasks. We do not apply these cache management techniques to level-C tasks as we expect the shared cache to provide reasonable average-case performance (which should be acceptable for SRT tasks). We assume that the last level of the shared cache is the one that is managed (*e.g.*, the L2 cache of our ARM platform). Furthermore, we assume that tasks of different criticality levels are *partitioned in* the cache, *i.e.*, they are allocated such that they do not share colors (support for such allocation is described in Section 3.6). The term "task" is used to refer to a level-B task in the rest of this section.

We begin by broadly describing color management strategies and our assumptions. First, we assume an inclusive write-back cache for which the OS can precisely control which cache sets and ways a task's data is loaded into. Second, we assume that the OS has access to hardware mechanisms that enable ways of the cache to be *locked* (and later *unlocked*) such hat data in a

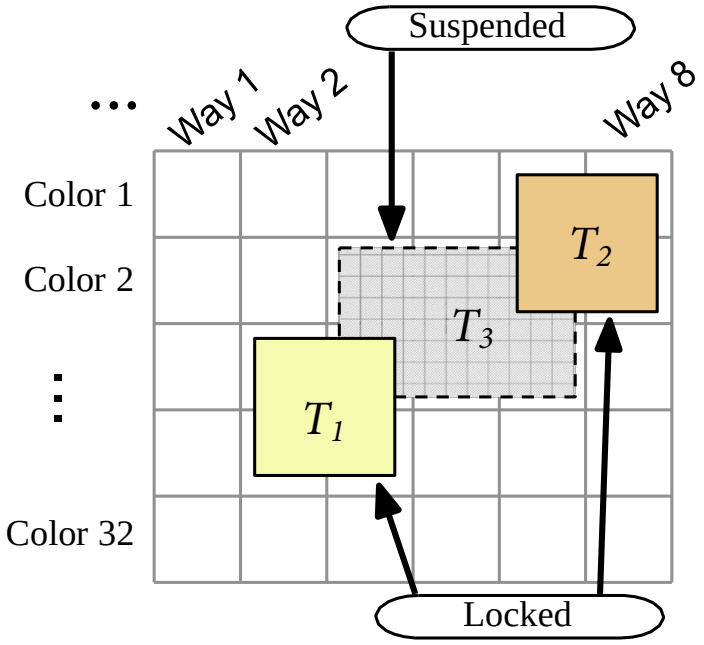


Figure 3: Color locking.

locked way is protected from being evicted (equivalently, the way is unavailable for allocation). Third, we assume that the physical memory pages, and therefore the color requirements of each task, have been previously assigned, and that tasks have been partitioned onto processors. Finally, we assume that the memory a task uses is pre-allocated before the task system begins execution, as is common in real-time systems. We discuss the realization of the first two points on our test platform in Section 3.6. In Section 4.2, we address the third requirement by describing methods to assign colors.

The effect of these assumptions is that once data is loaded into a cache line and it is locked, all subsequent memory accesses to that data will hit in the cache. With this property, uniprocessor timing analysis tools, which are much less pessimistic than multicore ones, can be employed to more accurately estimate WCETs for level-B tasks. Furthermore, observed WCETs will be improved as the number of cache misses is provably reduced.

Under these assumptions the cache can be treated as either a non-preemptive or a preemptive resource. This gives rise to four different classes of allocation policies, depending upon whether the processor or the cache is preemptive. We claim, however, that either both should be preemptive, or both should be non-preemptive. If the processor is non-preemptive and the cache is preemptive, then it is possible for a job to be scheduled while its needed cache colors have been preempted (*i.e.*, are not available for it to use). Alternatively, if the cache is non-preemptive and the processor is preemptive, then colors could be locked by a job that is later preempted and thus unable to use them. We therefore consider only the non-preemptive case, which we call *cache locking*, and the preemptive case, which we call *cache scheduling*.

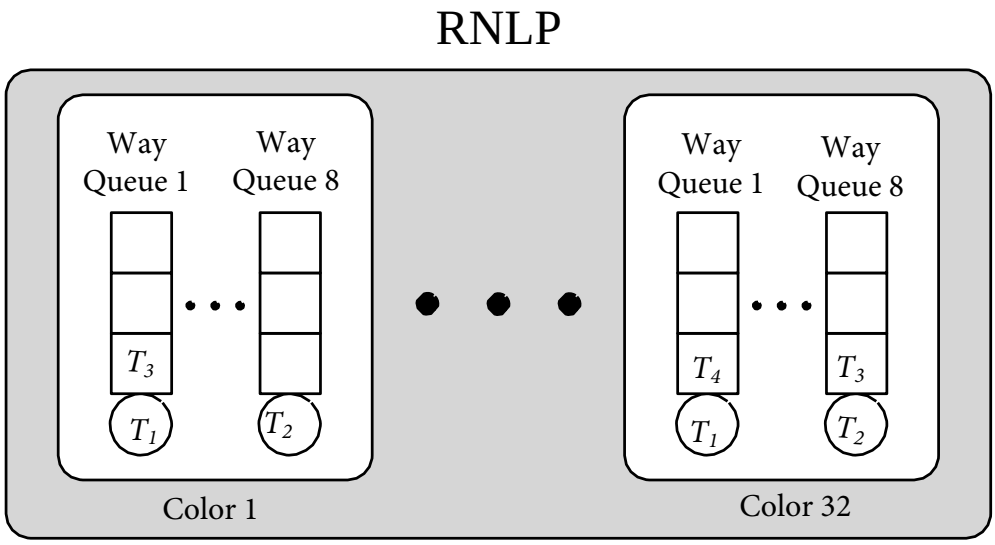


Figure 4: Illustration of the RNLP queue structure.

3.4.1 Cache Locking

Under cache locking, a job must hold a *color lock* for all of its needed colors before execution, and it does not release its color locks until the end of its execution, *i.e.*, its execution requirement is a critical section. This ensures cache isolation for each job during the entirety of its execution. This policy can be realized by using a multiprocessor real-time locking protocol to arbitrate access to colors and treating each job's execution time as a critical section.

For this purpose, we leverage the Real-Time Nested Locking Protocol (RNLP) [56], a recently developed multiprocessor real-time locking protocol that optimally supports the simultaneous locking of multiple resources. The RNLP controls access to all cache colors and their respective ways. For each way of each color, there is an associated FIFO-ordered *way queue* of jobs. This architecture is depicted in Figure 4. The head of each way queue is assumed to have acquired the associated way, though it does not execute until it has acquired all needed ways. A job J_i atomically requests all colors it requires before it can commence execution. For each color c from which J_i requests r_c ways, J_i is enqueued in the shortest r_c way queues associated with c. A job in a way queue that is either waiting for a resource or scheduled with its needed ways is considered non-preemptive. Therefore, no other jobs on the processor can be either scheduled or enqueued in the way queues. Because there can be at most m jobs total in all way queues, and because jobs enqueue in the shortest way queues, the maximum duration of blocking for all cache colors is $O(mr/k)$ where k is the number of ways available and r is the maximum number of ways per color requested by any job. Under s-oblivious analysis, a non-preemptive job is analytically treated as scheduled, even when it is not actually scheduled. This non-preemptive job can cause $O(mr/k)$ non-preemptive blocking for other jobs. Thus, the total duration of blocking is $O(mr/k)$.

Example 1. Consider a two-processor system, with two tasks on each processor, as depicted in Figure 5. For simplicity, assume that the cache is direct-mapped (*i.e.*, only one way per color).

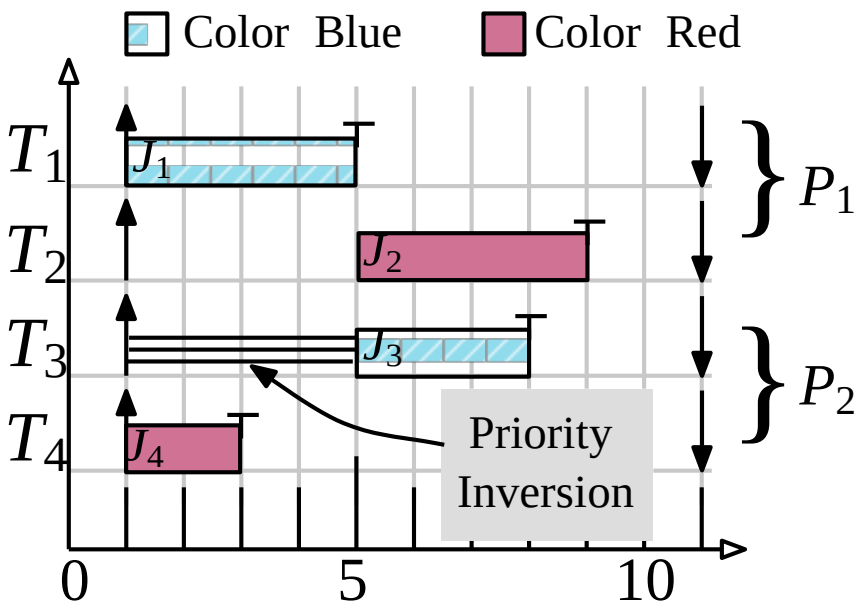


Figure 5: Example of the trade-off between priority inversion and improved WCET (explained in text).

Assume that jobs J_1 and J_2 (respectively, J_3 and J_4) are partitioned onto processor P_1 (respectively, P_2). Also, assume the lower-indexed jobs have higher priority. Let jobs J_1 and J_3 share blue (striped in the figure), and jobs J_2 and J_4 share red (solid). If all jobs are released synchronously on both processors, then without cache locking, both jobs J_1 and J_3 would be concurrently scheduled, and would conflict in the cache. However, as shown in Figure 5, J_1 is concurrently scheduled with J_4. These jobs do not conflict in the cache. When J_1 completes, J_2 and J_3 can execute without conflicting in the cache. In this example, there is a *priority inversion*, when job J_3 cannot run, and instead the lower-priority job J_4 executes. *Our goal is to show that the effect of such priority inversions is offset by the improved WCETs afforded by cache isolation.*

Period splitting. Non-preemptive processor scheduling, a byproduct of our locking protocol, can cause pi-blocking that can be detrimental to the schedulability of a task set.

Example 2. Consider a system where T_i has a higher priority than T_j. If $e_j \geq p_i$, then the system may be unschedulable because T_j might be non-preemptive throughout the entirety of T_i's period, causing T_j to miss its deadline.

The undesirable effects of non-preemptivity can be ameliorated by exploiting the fact that cache-related critical sections are not "true" critical sections: they can be preempted, albeit with an additional cost to reload evicted cache lines that are later required. Under s-oblivious analysis (recall from Section 3.3), non-preemptive blocking can be eliminated using a technique called *period splitting*. Under period splitting, each task's period is set to the shortest period in the system and its execution time is scaled accordingly. Thus, each job actually executes as a sequence of subjobs.

Example 3. Consider a system with two tasks, T_1 and T_2, where $e_1 = 1$, $p_1 = 3$, $e_2 = 6$, and

$p_2 = 9$. Under period splitting, T_2's period is reduced to match p_1, without altering its utilization. Thus, after period splitting, $e_2 = 2$ and $p_2 = 3$.

Since level B in MC^2 is simply periodic, period splitting ensures that subjobs are always re- leased in phase. Under this assumption, the RNLP guarantees that all s-oblivious pi-blocking is caused by tasks on remote processors.

Job splitting. Such remote blocking can clearly cause the system to be unschedulable. It is particularly detrimental if critical-section lengths are highly variant.

Example 4. Consider a system in which a color is shared between two tasks T_1 and T_2, where $e_1 = 1$ and $p_1 = e_2 = 16$. If T_1 blocks for 16 time units on T_2, then it will miss its deadline.

We can mitigate such detrimental blocking by breaking each task into multiple subtasks that are scheduled on the same processor and that all have the same period, but smaller execution times, such that all subtask utilizations sum to the original task's utilization. This is similar to inserting preemption points into jobs of a task, but is implemented by enforcing budgets. We call this technique *job splitting*. Note the difference between job and period splitting: under job splitting, tasks are broken into many subtasks with smaller utilizations, while under period splitting, a task's period is reduced, while maintaining its utilization.

Example 4 (cont'd). If T_2 is split into 16 subtasks, each with an execution cost of one, then worst-case blocking for T_1 is improved from 16 to one.

Job and period splitting, which can potentially be applied together, can reduce pi-blocking bounds by ameliorating the adverse effects of highly variant critical-section lengths. However, under either scheme, there is additional overhead for each subjob, such as scheduling decisions and reloading cache lines. Such overheads could be prohibitive if splitting is too "fine-grained."

3.4.2 Cache Scheduling

While non-preemptively scheduling tasks with respect to the cache maximizes reuse of cache lines within each job, it can cause adverse blocking, as we have seen. Alternatively, we can treat the cache as a *preemptive resource*, a technique we call *cache scheduling*. In this case, color replicas are preemptively "scheduled." When a task is scheduled with respect to a set of color replicas, it has exclusive access to those replicas and thus will not experience cache conflicts with tasks on remote processors. However, it may be preempted to allow a higher-priority task to access some replica. Similar to job and period slicing, such a preemption may force the preempted task to reload its WS. However, the cost of such reloads can be analyzed and incorporated into schedulability tests using existing techniques as described in [10, Chapter 3].

Example 5. Consider the two-processor schedule depicted in Figure 6, where each processor, P_1 and P_2, has two assigned tasks with utilization 1/4. Tasks T_1 and T_3 are defined by $(e_1, p_1) = (e_3, p_3) = (1, 4)$ and both share red and green. Task T_2 is defined by $(e_2, p_2) = (2, 8)$ and requires green only, while T_4 is defined by $(e_4, p_4) = (4, 16)$ and requires red only. RM priorities are applied to tasks on both the cache and their r espective processors. Thus, tasks T_1 and T_3 are

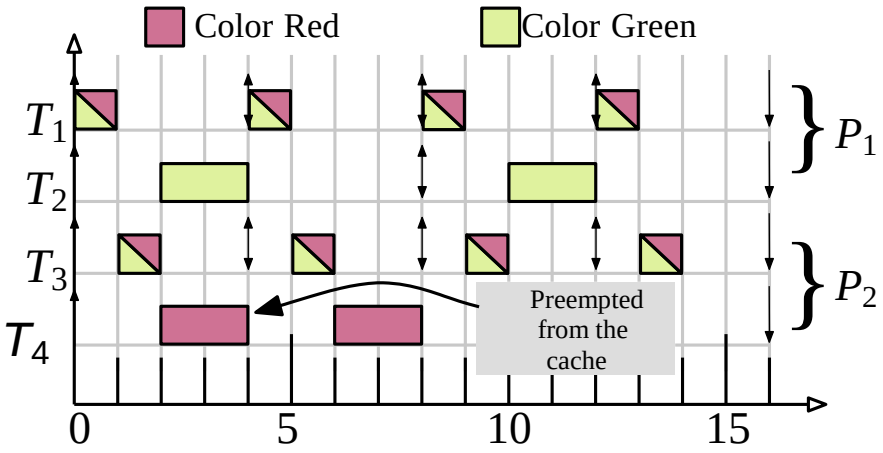


Figure 6: Preemptively scheduling the cache.

scheduled before T_2 and T_4. At time $t = 4$, T_4 is preempted by the higher-priority task T_1 on processor P_1.

We believe that by viewing cache colors as preemptive processors we expose a new scheduling problem. The problem is how to schedule and verify the schedulability of a task system in which each task T_i has an additional parameter, R_i, which gives a set of *processor requirements*. A processor requirement $(a, s) \in R_i$ specifies that for T_i to execute, it must be scheduled on a processors from the set of processors s.

Example 6. For a task T_i assigned to processor P_2 requiring four replicas of color blue, of which there are 16 ways, $B_1, \ldots, B_{16}$, we have $R_i = \{(1, \{P_2\}), (4, \{B_1, \ldots, B_{16}\})\}$.

This scheduling problem is a generalization of gang scheduling [30], and scheduling with processing set restrictions [35]. We leave this general problem for future work, but present more simplistic analysis by reducing the problem to a uniprocessor scheduling problem.

Reduction to uniprocessor analysis. Fundamental to our analysis is the observation that there are task systems for which the processor may not be the constrained resource—instead the cache may be overutilized, causing the task system to be unschedulable.

Example 7. Consider two tasks that share the same set of colors and have a total utilization greater than one. The tasks may be partitioned onto either one or two processors. In the former case, the processor is overutilized, and the task system is unschedulable. In the latter case, the tasks must run sequentially to ensure cache isolation, and thus the task system is unschedulable despite the additional processing capacity of the second processor. This is because the cache is the constrained resource, not the processors.

We next formalize the reduction to uniprocessor analysis mentioned above and demonstrate how the utilization of cache colors can be evaluated to check schedulability with respect to both cache colors and processors.

We define a binary relation for *direct contention D*, such that two tasks are related if they share

colors or a processor,

$$D = \{(T_i, T_j) \in \tau^2 \mid C_i \cap C_j \neq \varnothing \vee P(T_i) = P(T_j)\},$$

where C_i is the set of colors that task T_i requires, and $P(T_i)$ is the processor on which T_i is partitioned. Let D^+ be the transitive closure of direct contention, D. We define each equivalence class in D^+ to be a logical *cache processor*. By definition, all tasks on the same physical processor must be on the same cache processor, and thus the number of cache processors is at most m.
We evaluate the schedulability of each cache processor as a uniprocessor, and apply known schedulability tests. In MC^2, which uses RM priorities on each physical processor and simply periodic periods, the schedulability test for a physical processor P_j is $\sum_{T_i \text{ on } P_j} u_i \geq 1$. We can also schedule the cache using RM priorities, and allow new jobs of high-priority tasks to preempt low-priority tasks with respect to the cache. Thus, the resulting schedulability test is the same: the total utilization of each cache processor is at most one. We do not need to explicitly evaluate the schedulability of the physical processors, since the set of tasks on each physical processor is a subset of the tasks on the respective cache processor. Thus, if each cache processor is schedulable, then each physical processor is as well.

Example 5 (cont'd). Because tasks on processors P_1 and P_2 share the same set of colors (red and green), they form a single cache processor. Since the utilization of the four tasks is one, both the cache and the processors are schedulable. If all tasks required both red and green, then the tasks would necessarily be serialized as if they were on a single processor (assuming cache isolation is required). However, because tasks T_2 and T_4 do not share the same colors, they can execute concurrently, as depicted in Figure 6.

Note that this schedulability condition is only sufficient and may be pessimistic: it may be possible for tasks to run concurrently on different physical processors if they require disjoint colors. This pessimism can be avoided in two ways: through tighter analysis of the aforementioned more general scheduling problem, or by assigning colors to tasks in a way that reflects our uniprocessor analysis, as discussed later.

3.5 Related Work

Prior work exists on cache management that is similar to ours in some respects. An approach called *cache lockdown* (not to be confused with our use of the term "locking") has been proposed wherein designated cache lines are "locked down" in the cache so that they cannot be evicted [16]. Similarly, an approach called *cache partitioning* has been proposed that attempts to mitigate the impact of cache conflicts by allocating sections (or partitions) of the cache to specific tasks (see [32] for an overview). Cache partitioning can be done automatically by the compiler [47], but the source code of programs must be available for compilation and large portions of memory must be allocated as padding to achieve the desired code and data placement. To remedy this, partitioning at the OS level was proposed [36]. This approach can be applied dynamically, transparently, and without access to application source code. However, it may be difficult to size partitions so that the cache is efficiently utilized from a system-wide perspective.

In general, the problems of optimally assigning tasks to processors and colors to tasks are both NP-hard in the strong sense [14], though suboptimal heuristic-based algorithms have been proposed and evaluated on different hardware platforms [26, 48, 50]. Cache partitioning is actually a special case of our cache locking approach in which inter-task cache conflicts are entirely eliminated, and hence the usage of a locking protocol is obviated. However, our generalization allows more flexibility in determining color assignments and can enable greater dynamism at runtime.

A system execution model called PREM [51] has been proposed that takes an approach similar to ours. Specifically, PREM uses scheduling to reduce or eliminate contention for shared resource accesses, including main memory. However, unlike our proposed cache management techniques, PREM is restricted to single-core systems. An extension to PREM [61] examines memory-centric Time Division Multiple Access (TDMA)-based scheduling on multicore processors, but assumes no shared resources among cores except memory—each core's last-level cache is private.

In work on timing analysis, methods pertaining to memory hierarchies have been proposed (*e.g.*, see [34, 52] and the references therein). Related hardware-based techniques include *cache bypass* [28], which reduces cache conflicts by exploiting special hardware instructions, and methods for making multicore platforms more predictable at the processor [49] and interconnect levels [3, 27, 53, 54]. In contrast to this work, our approaches are software-based.

3.6 Implementation

We implemented the cache management techniques described above in an MC^2 prototype that was itself implemented within a real-time extension of Linux called $LITMUS^{RT}$ (**LI**nux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime systems) [1]. $LITMUS^{RT}$ was developed previously by our group under ARO and NSF support. $LITMUS^{RT}$ extends Linux (currently, version 3.0) by allowing different (multiprocessor) scheduling algorithms and synchronization protocols to be linked as plug-in components. Being based on Linux, $LITMUS^{RT}$ is not a real-time OS (RTOS) that could conceivably be used in safety critical domains. However, we chose it as our development platform because it is open source. This choice of platform should not be a limitation: any resource allocation principles identified in our work should be easily transferable to other RTOSs.

In developing our implementation, we restricted attention to tasks that are independent (no shared resources other than cache lines), have only memory-resident pages, and that do not share pages with each other. We used the ARM machine described in Section 3.3 as our development platform. All page coloring was done with respect to the last level of cache, which on this machine is its L2. In prior work, we found that level-B schedulability is greatly improved if one CPU is designated as a *release master* that processes all job-release interrupts and is not assigned any level-B tasks [29]. We employed a release master in our current prototype, so level-B tasks are actually only scheduled on three CPUs (level-C tasks can execute on the release master).

As noted earlier, we implemented coloring with respect to tasks' data pages. We leave the coloring of other areas of memory as future work. We believe this is reasonable for a first prototype, as our implemented tasks have a small per-job code footprint that does not make any system calls and operate only on the colored memory they allocate. We implemented a memory allocation function similar to malloc that modifies the page tables of the backing user process for each task to map pages with the proper colors into its contiguous virtual address space.

Controlling eviction. When discussing various page-coloring-oriented cache management

techniques in Section 3.3, we assumed that the OS can precisely control the cache sets and ways that a task's data is loaded into. In our implementation, this functionality is achieved by exploiting cache lockdown using a clever method proposed by Mancuso, *et al.* [43]. As noted earlier, *cache lockdown* allows certain ways to be marked as unavailable for allocation (or locked down) such that the contents of a locked way cannot be evicted.

The approach in [43] utilizes a variant of cache lockdown called *Lockdown by Master* (*LbM*), where each CPU P_q has access to a per-CPU lockdown register x_q such that bit $\boldsymbol{i}$ in x_q is zero if allocation can occur in way $\boldsymbol{i}$ for memory references from CPU P_q, and one otherwise. LbM is supported on our ARM platform. As noted by the authors of [43], setting all but one bit of $\boldsymbol{r}_c$ to one pigeonholes memory requests from CPU P_q to be allocated in a specific cache way. By applying this idea, memory can be prefetched by reading it in a "prefetch" loop such that the loop code occupies at most one cache line and is cache-line aligned. During prefetching, CPU-local interrupts must be disabled to avoid interrupt-related cache pollution. To ensure that each prefetched cache line is read into the proper way, none of the memory to be prefetched can be cached elsewhere. Also, if a CPU is prefetching into way *w*, then the other CPUs cannot have *w* unlocked unless they access memory that is of a different color.

4 RESULTS AND DISCUSSION

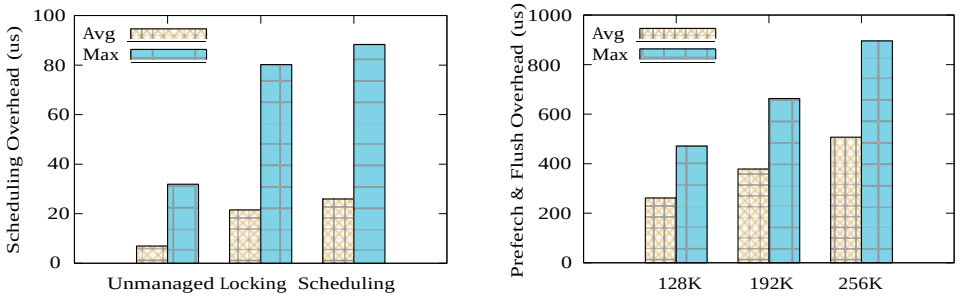
In this section, we describe the results of an evaluation that we conducted to assess the efficacy of our cache management framework. In Section 4.1, we describe our evaluation methodology. In Section 4.2, we describe our experimental results.

4.1 Evaluation

We evaluated our implementation by measuring system overheads (including cache prefetching and flushing costs) and observed WCET improvements. To obtain these measurements, we traced the behavior of task sets with varying task counts, where periods and utilizations were generated uniformly from {25, 50, 100, 200}ms and [0.01, 0.05], respectively. Level-B tasks were assigned to cores using the worst-fit heuristic. (Any task set that could not be so assigned was discarded.) Cache prefetching was turned off. Each task was defined via a per-job code sequence in which it reads its WS in a random order one or more times.

Scaling factors. A job executed under cache locking or cache scheduling does not suffer cache misses and therefore completes execution earlier than it would in a system without cache management. We use *scaling factors* to denote the ratio of per-job execution times without and with cache management, *e.g.*, if cache scheduling reduces a task's per-job execution time from 10 ms to 2 ms, then this task has a scaling factor of five. The worst-case scaling factors on our test platform were on the order of 3.5 to 4.5, while average-case scaling factors were between 3.3 and 3.9.

In Section 4.2, we use these scaling factors to compare schedulability with and without cache management in the presence of system overheads. While it might be preferable to use WCETs



(a) Scheduling overhead. (b) Prefetch & flush overhead.

Figure 7: Each graph shows average-case (orange, textured) and worst-case (blue, solid) measurements. (a) Scheduling overheads. (b) Prefetch and flush overheads for various WSSs.

predicted by timing analysis tools in such comparisons, adequate tools for multicore platforms do not yet exist. Also, note that observed WCETs lower bound predicted ones (if the prediction is safe). Thus, observed values give some indication of how a tool might perform given varying degrees of information about cross-core cache interactions.

System overheads. Cache management increases scheduling costs and consequently the duration of OS overheads interrupting task execution. Additional overhead arises under cache locking due to locking-protocol overhead, and under cache scheduling due to added complexity for maintaining cache processor state. Figure 7(a) depicts scheduling overheads (*i.e.*, the time taken by the OS to make a scheduling decision) for level B assuming no cache management (*Unmanaged*), cache locking (*Locking*), and cache scheduling (*Scheduling*). Observe that worst-case overheads were increased by around 60 μs for cache scheduling and 50 μs for cache locking.

The cache lockdown prefetching operation must also be considered. A task will not begin execution until its assigned processor has cached the entirety of its WS. In Figure 7(b), we show this total prefetching cost for different WSSs. The worst-case cost varied from 400 μs to 900 μs in our experiments, depending on WSS. With respect to schedulability, these costs and higher scheduling overheads are offset by lower WCETs as we show next.

4.2 Schedulability Study

In this section, we evaluate the utility of our proposed cache management techniques from a schedulability perspective, with measured overheads considered, and examine the tradeoff between improved WCETs enabled by our techniques and any utilization loss due to cache management.

Utilization scaling. To evaluate the schedulability of a task system using cache management, we scale the level-B utilization of tasks scheduled without cache management by a scaling factor commensurate with those observed in Section 4.1. This utilization scaling theoretically allows a scheduler using our cache management techniques to schedule task systems that, unmanaged, contain tasks with level-B utilizations greater than one or have a total level-B utilization greater than the number of processors. For example, using our cache management techniques, it may be

possible to schedule on four processors a task system with a total unmanaged level-B utilization of five and that contains a task with an unmanaged utilization of 1.5. These observations motivate the experimental design of our schedulability study, as is discussed next.

We evaluated the schedulability of randomly generated task systems having equal level-B and -C subsystem utilizations. That is, for each system utilization x, we generated task systems where the sum of all level-B tasks' level-B utilizations was equal to x, and the sum of all level-B and -C tasks' level-C utilizations together also summed to x (recall that in mixed-criticality scheduling, a task has an execution cost, and hence utilization, for each criticality level). The level-C utilization of each level-B task was assumed to be 10% of its level-B utilization. We considered task systems with utilizations without cache management of $\{1.0, 1.1, \ldots, 10.0\}$ with respect to the previously described system (four processors, 1MB 8-way set associative cache, with 512 KB allocated to each criticality level). Both the level-B utilizations of level-B tasks and the level-C utilizations of level-C tasks were uniformly distributed over [0.1,0.4] or [0.5,0.9], though in all graphs presented herein, which depict relevant trends, the former utilization distribution is assumed. We note that greater per-task utilizations could be supported using cache management (*e.g.*, 1.5). However, we did not consider such systems in our evaluations as there is no basis for comparison with a system with an unmanaged cache (which cannot schedule tasks with utilizations greater than 1.0).

Schedulability was determined by evaluating the generated task systems using the level-B and -C schedulability conditions given in [46], with overheads factored in using the techniques described in [10, Chapter 3]. Worst-case (average-case) overheads were assumed for level B (C), as level B (C) is HRT (SRT). Comparisons between systems with and without cache management were performed by generating task systems for an unmanaged system, and then scaling task execution times by scaling factors commensurate with those observed in Section 4.1.

Coloring schemes. We investigated two heuristics for assigning colors to level-B tasks. Each was designed to reduce cross-core color contention, since such contention is detrimental to both cache locking and cache scheduling. Both heuristics function similarly and are applied assuming that tasks have been previously assigned to physical processors and that physical memory pages are unconstrained (conceptually, there are infinitely many physical memory pages available for each color—note that our test platform has 2^{13} pages per color). Letting S denote the size of the cache and W denote the maximum WSS, we divide the cache into $\lfloor S/W \rfloor$ "bins", each of size W. Under *way-first* (*color-first*) *allocation*, the number of ways (colors) in each bin is maximized. For example, a bin half the size of an 8-way, 32-color cache has 8 ways of 16 colors under way-first allocation, and 4 ways of 16 colors under color-first allocation. Under either heuristic, processors are packed into the obtained bins using the worst-fit heuristic, and each task is assigned colors to satisfy its WSS from the set of colors assigned to the processor on which it is partitioned. Note that if $W \leq S/m$, then our heuristics partition the cache.

These heuristic are more flexible than cache partitioning, as they allow colors to be shared across processors. However, the problem of assigning colors to tasks is generally intractable and may be particularly hard if the number of physical memory pages per color is severely constrained. In many embedded systems, such constraints may exist due to SWaP requirements, the need to support separate processing modes, *etc.* Nonetheless, the experiments below demonstrate that

if a reasonable color assignment can be found, then schedulability-related impacts can be dramatic. We plan to further investigate color assignment strategies in the future to widen the applicability of our work.

Schedulability. We now discuss several schedulability graphs involving randomly generated task systems. A cache management scheme's *schedulability* is specified with respect to a set *S* of generated task systems and is defined as the fraction of *S* deemed schedulable under that scheme. We present level-B (HRT) and level-C (SRT) schedulability graphs in Figure 8. In the level-B graphs, schedulability is plotted versus total level-B utilization *prior to scaling*. In the level-C graphs, it is plotted versus total level-B and -C utilization assuming level-C execution costs for both. With respect to the level-B scheduling policy, we denote ordinary partitioned RM (P-RM) scheduling with an unmanaged cache (*i.e.*, no cache locking or scheduling) as *UM*, cache locking as *CL*, cache scheduling as *CS*, and cache locking with period splitting as *CL-PS*. We do not present graphs for job splitting as it was always inferior to period splitting alone. To prevent interference between tasks of different criticality levels, level-B and -C tasks are each assigned half of the available cache, so a WSS of 256K corresponds to half of the available level-B cache. Our major observations are as follows.

Observation 1. In all observed cases, at least one of the proposed cache management solutions (CL, CS, or CL-PS) offered improved level-B schedulability, often scheduling task systems with almost 50% greater processor utilization than an unmanaged P-RM level-B subsystem (UM).

This observation is corroborated by insets (a)-(e) of Figure 8, which depict level-B schedulability. For example, in Figure 8(a), with a scaling factor of three, which is less than we observed in practice, cache scheduling (CS) could schedule task systems with a level-B utilization of 50% more than that of an unmanaged P-RM system (UM). By scaling level-B tasks' level-B execution times down (to account for fewer cache misses due to cache management), we are able to schedule task systems that previously would have been unschedulable because they would have overutilized the available processing cores. This effect is even more pronounced with larger scaling factors. These results show that the benefits of cache management techniques may outweigh the increased overheads they entail.

Observation 2. For some system configurations, unmanaged P-RM (UM) offered improved level-B schedulability over one or more cache management techniques. This suggests that the cache can be managed *improperly*.

In the level-B schedulability results presented in insets (a)-(d) of Figure 8, unmanaged P-RM (UM) outperforms cache locking (CL) in many cases and in Figure 8(d), unmanaged P-RM (UM) outperforms cache scheduling (CS). However, as described in Observation 1, at least one of our cache management solutions outperforms unmanaged P-RM (UM). Thus, a system designer must evaluate their task system to decide which of our cache management solutions is best.

Observation 3. In all observed cases, period splitting improved level-B schedulability under cache locking.

This can be seen in insets (a)-(e) of Figure 8. Henceforth, when we refer to cache locking, we

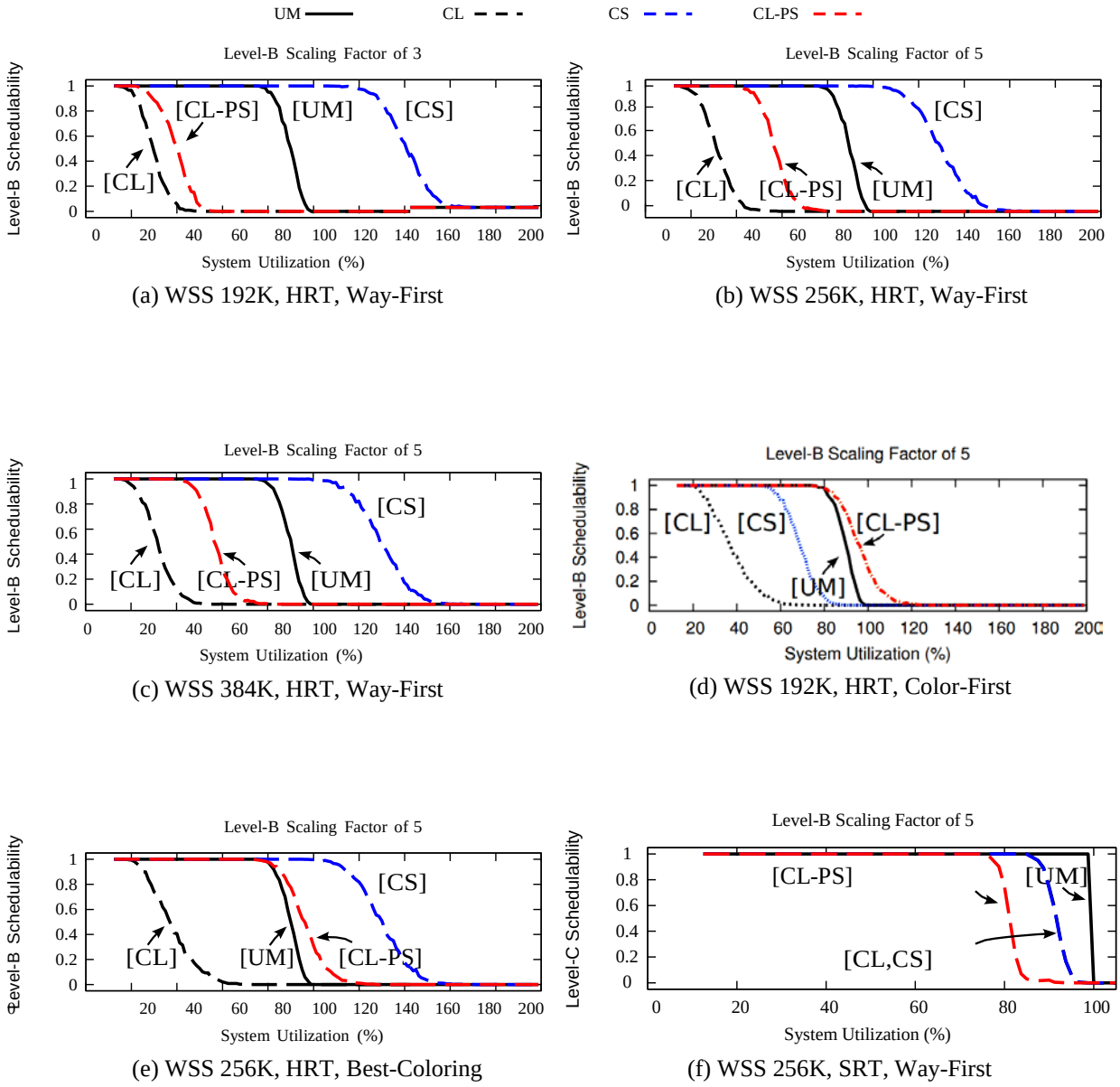


Figure 8: Insets (a)-(d) compare the proposed cache management approaches in terms of level-B schedulability for different scaling factors, WSSs, and coloring algorithms. Inset (e) compares the schedulability of the proposed cache management techniques using the coloring algorithm that provides the best schedulability for each technique, respectively. Inset (f) depicts level-C schedulability.

assume period splitting (CL-PS) is used.

Observation 4. Under color-first allocation, cache locking (CL-PS) offered the best level-B schedulability, while under way-first allocation, cache scheduling offered the best level-B schedulability. This suggests that color assignment has significant impact on level-B schedulability.

Comparing level-B schedulability using way-first and color-first allocation in insets (b) and (d) of Figure 8, respectively, we observe that cache locking (CL-PS) performs much better under color-first allocation. This is because the blocking bound for cache locking is $O(mr/k)$ where r is the maximum number of ways of a color required, which is minimized under color-first allocation, and k is the total number of ways. In contrast, cache scheduling has much better level-B schedulability under way-first allocation, but performs comparatively worse under color-first allocation. This is because of the pessimism of our analysis, which assumes that only one task executes with a color at a time, regardless of the number of ways required (recall from Section 3.4 that tighter analysis for cache scheduling is an open problem). This pessimism is particularly detrimental to level-B schedulability using color-first allocation. However, under way-first allocation, tasks use all of the ways of comparatively fewer colors, which in turn reduces the number of cache processors, and improves schedulability. These observations demonstrate the impact that the coloring algorithms have on level-B schedulability.

Observation 5. The overheads associated with our cache-management techniques only minimally impacted level C.

Our techniques have a profound impact at level B. However, as can be seen in Figure 8(f), schedulability at level C is decreased by a comparatively small amount. While this offset means we can schedule no additional level-C work, we could execute many more high-criticality level-B tasks in lieu of lower-criticality level-C tasks.

Observation 6. In all observed cases, cache scheduling under way-first allocation provided better level-B schedulability than any other cache management technique under either way-first or color-first allocation.

This can be seen in insets (a)-(e) of Figure 8. Under color-first allocation, as in Figure 8(d), level-B schedulability under cache scheduling (CS) was less than that of cache locking (CL- PS). However, in all cases in which cache locking (CL-PS) outperforms cache scheduling using color-first allocation, level-B schedulability using cache scheduling under way-first allocation was greater. This can be seen in Figure 8(e), which shows for each technique the best level-B schedulability seen under either coloring scheme.

Cache scheduling also has practical benefits over cache locking. Cache scheduling is easier to implement, being little more than a multiprocessor RM algorithm with an extra “mapping” step, while cache locking uses both partitioned RM and a locking protocol. Additionally, cache scheduling is more flexible than cache locking (which uses period splitting). For example, it can be more easily applied to schedule non-periodic task systems. Thus, unless a system is unable to use way-first page allocation, cache scheduling is the recommended cache-management technique.

5 CONCLUSIONS

In this project, we devised several techniques for managing shared caches on multicore systems within the MC^2 mixed-criticality scheduling framework. We also experimentally evaluated an implementation of these techniques on a quad-core ARM machine. These experiments indicate that proper shared cache management can lessen WCETs and positively impact schedulability despite increased system overheads. In fact, in our experiments, task systems could be scheduled successfully with cache management that had total utilizations exceeding the capacity of our test platform by 50% or greater without cache management. The development of these cache management techniques as well as MC^2 was motivated by ongoing work with colleagues at NGC on defining useful real-time resource allocation infrastructure for future UAVs. These UAVs represent an interesting challenge problem in work on cyber-physical systems. Much of the work included in this document appears in [58], which will receive an Outstanding Paper Award at ECRTS 2013.

Future work. This work opens many avenues for future research, which we plan to pursue, ranging from systems-level issues to theoretical scheduling problems. For example, we plan to explore cache management techniques for shared libraries, dynamic memory, and jobs with WSSs larger than the cache. We also plan to evaluate, from the perspective of rigorous timing analysis, the improvements to calculated WCETs made possible by these cache management techniques. Additionally, we are interested in developing scheduling algorithms, schedulability tests, and resource allocation techniques that can support both greater system utilizations (with respect to both the processors as well as the cache) and more than two criticality levels.

Other research results. In addition to shared cache management, we have investigated a variety of other topics that are related to the research covered above. Some of this work was co-funded by other agencies (namely NSF, ARO, and AFOSR). Such topics include the following.

• **GPU-enabled multicore real-time systems:** Graphics processing units (GPUs) can be used in real-time systems to significantly speed up computations that are parallelizable. However, predictably managing GPUs so that real-time constraints can be ensured is challenging. We have shown that the needed predictability can be attained by using multiprocessor real-time scheduling and synchronization techniques to control GPU accesses [20, 22, 23, 25].

• **Optimal real-time multiprocessor locking protocols:** Real-time applications require locking protocols for controlling accesses to shared resources that are amenable to real-time analysis. In recent years, the advent of multicore technologies has led to a renewed interest in real-time locking protocols for multiprocessors. In a series of papers, we have presented a variety of such protocols, for both mutual exclusion (mutex) and reader/writer synchronization [11, 12, 13, 21, 56, 57]. (Two of these papers won best paper awards [11, 12], at RTSS 2010 and EMSOFT 2011, respectively.) Our protocols are the first to be asympotitically optimal despite over two decades of research on this topic. As seen herein, we leveraged this work in designing our synchronization-oriented cache management framework.

• **SRT systems with self-suspensions and precedence constraints:** In practice, real-time tasks may interact with each other or external devices. Such interactions can result in task

suspensions and precedence constraints, both of which may complicate real-time schedulability analysis. We have produced several analysis techniques for dealing with such complications when computing tardiness bounds for SRT tasks [37, 38, 39, 40, 41].

• **Middleware and RTOS issues:** In this project, we have assumed that mixed-criticality resource allocation techniques should be implemented within the RTOS. However, through interactions with employees of RTOS companies embedded at NGC, we learned that a complete RTOS implementation may be unlikely due to market forces and other reasons. Thus, an appropriate factoring of functionality between the RTOS and middleware is needed. We have published two papers in which such issues are considered [44, 45]. We have also published several papers in which implementation-oriented tradeoffs involving real-time schedulers (including MC^2) have been investigated [8, 9, 24, 29, 31].

6 BIBLIOGRAPHY

[1] LITMUS^{RT} home page. http://www.litmus-rt.org/.

[2] U.S. Air Force Technology Horizons. http://www.af.mil/information/technologyhorizons.asp.

[3] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 251–256, September 2007.

[4] T. Baker. The cyclic executive model and Ada. *The Journal of Real-Time Systems*, 1:120–129, 1989.

[5] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In Sang H. Son, Insup Lee, and Joseph Y. Leung, editors, *Handbook of Real-Time and Embedded Systems*. Chapman Hall/CRC, Boca Raton, Florida, 2007.

[6] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 145–154, July 2012.

[7] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 13–22, April 2010.

[8] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *Proceedings of the IEEE 31st Real-Time Systems Symposium*, pages 14–24, December 2010.

[9] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 125–135, July 2011.

[10] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

[11] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60, December 2010.

[12] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and *k*-exclusion locks. In *Proceedings of the ACM International Conference on Embedded Software*, pages 69–78, October 2011.

[13] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, pages 1–66, 2012. Digital Object Identifier 10.1007/s10617-012-9090-1.

[14] B. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110, August 2008.

[15] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256, July 2007.

[16] M. Campoy, A.P. Ivars, and J.V.B. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of the IEEE Real-Time Embedded Systems Workshop*, pages 1–6, December 2001.

[17] W. Dahm. Certifiable trust in autonomy through V&V. Keynote talk, 2012 S5 Symposium, Fairborn, OH, June 2012. https://www.signup4.net/public/ap.aspx?EID=SAFE23E& OID=110.

[18] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, December 2005.

[19] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 135–144, July 2012.

[20] G. Elliott and J. Anderson. Real-time multiprocessor systems with GPUs. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, pages 197–206, November 2010.

[21] G. Elliott and J. Anderson. An optimal *k*-exclusion real-time locking protocol motivated by multi-GPU systems. In *Proceedings of the 19th International Conference on Real-Time and Network Systems*, pages 15–24, September 2011.

[22] G. Elliott and J. Anderson. Real-world constraints of GPUs in real-time systems. In *Proceedings of the First International Workshop on Cyber-Physical Systems, Networks, and Applications*, pages 48–54, August 2011.

[23] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[24] G. Elliott and J. Anderson. The limitations of fixed-priority interrupt handling in PRE- EMPT RT and alternative approaches. In *Proceedings of the 14th Real-Time Linux Workshop*, pages 149–155, October 2012.

[25] G. Elliott and J. Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 267–276, July 2012.

[26] M. Fernàndez, R. Gioiosa, E. Quin˜ ones, L. Fossati, and F. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In *Proceedings of the International Conference on Embedded Software*, pages 175–184, October 2012.

[27] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22:414–421, 2005.

[28] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 68 – 77, December 2009.

[29] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multi-core mixed-criticality systems. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.

[30] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 459–468, December 2009.

[31] C. Kenna, J. Herman, B. Brandenburg, A. Mills, and J. Anderson. Soft real-time on multi-processors: Are analysis-based schedulers really worth it? In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 93–103, December 2011.

[32] D. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium RTSS '89*, pages 229–237, December 1989.

[33] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 413–422, December 2007.

[34] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In *9th Int'l Workshop on WCET Analysis*, June 2009.

[35] J.Y.-T. Leung and C.-L. Li. Scheduling with processing set restrictions: A survey. *International Journal of Production Economics*, 116:251–262, December 2008.

[36] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–224, June 1997.

[37] C. Liu and J. Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 3–13, December 2010.

[38] C. Liu and J. Anderson. Supporting graph-based real-time applications in distributed systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 143–152, August 2011.

[39] C. Liu and J. Anderson. An $O(m)$ analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 373–382, December 2012.

[40] C. Liu and J. Anderson. Supporting soft real-time parallel applications on multicore processors. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 114–123, August 2012.

[41] C. Liu and J. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 271-281, July 2013.

[42] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, January 1973.

[43] R. Mancuso, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time colored lockdown for cache-based multi-core architectures. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, April 2013.

[44] M. Mollison and J. Anderson. Virtual real-time scheduling. In *Proceedings of the Seventh International Workshop on Operating Systems Platforms for Embedded Real-Time*, pages 33–40, July 2011.

[45] M. Mollison and J. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 283–292, April 2013.

[46] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed criticality real-time scheduling for multicore systems. In *Proceedings of the 7th IEEE International Conference on Embedded Software and Systems*, pages 1864–1871, June 2010.

[47] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995.

[48] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Proceedings of the 9th European Dependable Computing Conference*, pages 132–143, May 2012.

[49] M. Paolieri, E. Quiñones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 57–68, June 2009.

[50] M. Paolieri, E. Quin˜ ones, F. Cazorla, R. Davis, and M. Valero. IA^3: An interference aware allocation algorithm for multicore hard real-time systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 280–290, April 2011.

[51] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *Proceedings of the 17th IEEE Real- Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011.

[52] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, April 2006.

[53] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108, October 2011.

[54] R. Stefan, A. Molnos, A. Ambrose, and K. Goossens. A TDM NoC supporting QoS, multicast, and fast connection set-up. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1283–1288, March 2012.

[55] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239–243, December 2007.

[56] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 223–232, July 2012.

[57] B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289, August 2012.

[58] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157-167, July 2013.

[59] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *The Journal of Real-Time Systems*, 21(3):241–268, 2001.

[60] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem — Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[61] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48:681–715, 2012.

7 LIST OF ACRONYMS

CASs	Complex Adaptive Systems
CL	Cache Locking
CS	Cache Scheduling
EDF	Earliest Deadline First
ECRTS	Euromicro Technical Committee on Real-Time Systems
FIFO	First In First Out
J-UCAS	Joint Unmanned Combat Air System
G-EDF	Global EDF
HRT	Hard Real-Time
LbM	Lockdown by Master
LITMUS^{RT}	LInux Testbed for Multiprocessor Scheduling in Real-Time systems
MC^2	Mixed-Criticality on Multi-Core
NGC	Northrop Grumman Corp
N-UCAS	Navy's Unmanned Combat Air System
OS	Operating System
P-EDF	Partitioned EDF
P-RM	Partitioned Rate Monotonic
PI-Blocking	Priority Inversion Blocking
PREM	PRedictable Execution Model
PS	Period Splitting
RM	Real-Time Nested Locking Protocol
RTOS	Real-Time Operating System
S-Aware	Suspension-Aware
SRT	Soft Real-Time
SWaP	Size, Weight, and Power
TDMA	Time Division Multiple Access
UAV	Unmanned Air vehicle
WCET	Worst-Case Execution Time
WS	Working Set
WWS	Working Set Size